Big data and urban mobility

Antònia Tugores**, Pere Colet

Instituto de Física Interdisciplinar y Sistemas Complejos, IFISC(UIB-CSIC)

Abstract. Data sources have been evolving the last decades and nowadays a huge amount of information is available through web navigation. One of the new sources of information are social networks and they provide large datasets to be studied. In this context we are using geolocalized tweets as a source of information for mobility patterns. Data retrieval is achieved through public APIs and smart users selection. Besides, efficient data storage and fast access to the data is a challenging task for which we rely on NoSQL technology.

Keywords: Data storage, data management, big data, NoSQL, distributed database

1 Introduction

The amount of data available electronically has been exploding in the last years. In parallel, the world's technological capacity to store information has increased unexpectedly. As of 2012, every day 2.5 quintillion (2.5×10^{18}) bytes of data were created, and 90% of the data in the world today has been created in the last two years alone. This data comes from everywhere: sensors used to gather climate information, social networks, digital pictures and videos, transaction records, cell phone calls and GPS signals, genomics or complex physics simulations to name just a few. Capturing, curating, storing, searching, sharing, transfering, analysing, and visualizating those large data sets, or big data, are becoming key basis for studying the society.

Taking into account the 3Vs model [1][2] (high volume, high velocity and/or high variety information), the data we are working with, obtained from Twitter [3] social network, fits in the definition of big data. Although it has a low information density, its huge volume allows to infer laws and, in our case, it may be helpful to study commuting and human mobility patterns.

The objective of this paper is to discuss efficient ways to retrieve, store and manage large amounts of data from social networks such as Twitter to study human mobility patterns. The paper is organized as follows, data retrieval is described in section 2, section 3 summarizes current databases and explains our decision to use MongoDB. Section 4 describes the database configuration we have implemented, and finally, concluding remarks are given in section 5.

** e-mail of corresponding author: antonia@ifisc.uib-csic.es

2 Data retrieval

In order to study the mobility in some big cities (mainly London, Barcelona and Zurich) we are using geolocalized tweets. One can get 1% of all the tweets by using the stream API provided by Twitter, but in this case, less than 12 % of the collected tweets are geolocalized. As the geolocalized tweets are distributed all over the world, only a small fraction of them are located in the cities we are focusing in. As a consequence the network of users that can be constructed is smaller than desired.

To solve this issue after some months of data recollection we identified the users that have tweets geolocalized in the cities considered here. Then, in addition to the stream data, we download the Twitter timeline of these specific users.

2.1 Stream

We use the Public Stream that offers samples of the public data flowing through Twitter and it is suitable for data mining.

Twitter allows for applications to establish a connection to the streaming endpoint and through this connection a random sample of 1% of the tweets can be downloaded. This avoids the limitations imposed by Representational State Transfer (REST) APIs. The only limitation is that each account can create only one standing connection to the public endpoints, and connecting to a public stream more than once with the same credentials causes the oldest connection to be disconnected. In the same way, IPs of clients that make excessive connection attempts run the risk of being automatically banned.

One of the particularities of the Streaming API is that messages are not delivered in the same precise order as they were generated. In particular messages can be slightly shifted in time and it is also possible that deleted messages are received before the original tweet. This is not critical for the case considered here because we are interested at slower time scales (from minutes to hours) and therefore we do not need to have an exact timing and order of the messages.

Twitter streaming API requires keeping a persistent HTTP connection permanently open, and, by using listeners, the process that opens the connection should perform all parsing, filtering, and aggregation needed before storing the result. In our case, we store the tweets we download in the same form as they are received while deleted tweets have to be modified since they have a different structure. To facilitate data search and manipulation we use the tweet id as one of the indices of the database.

2.2 Users selection

In order to increase the number of geolocalized tweets in the cities to be studied, we identify users that on a specific period of time have posted at least one geolocalized tweet in one of these cities and then we download its timeline (that is the tweets the user has posted with some limitations).

First of all, we identify the tweets geolocalized in the three cities. This can be done though geoNear [6] MongoDB command. With this command we can specify a point for which the geospatial query returns the closest documents not exceeding a desired distance (radius) from the given point. Some databases limit the size of the results returned by a query. In the case of MongoDB this limitation is of 16MB [7] if not using GridFS [8]. In order to avoid exceeding this limitation we use a value for the radius of exploration of one mile and in order to cover all space in the city we make use of a fine grained mesh in which the points are separated by one mile.

2.3 Geolocalized data and users network

Twitter REST API has number of queries per time limitations, as for the methods we use the limit is 1 query every 15s. That makes a total of 43200 queries per month. And to avoid IP banning we try to keep away from the maximum.

For all the users in the selected group, we continuously get the timeline from the last tweet we collected and store the id of the last tweet we retrieve.

To retrieve the users network we get the list of uids collected and for each one, we get the followers and friends uids at the moment of running the query. In order to see the network evolution, the process is continuously running to get the network in different moments of time.

3 SQL vs NoSQL

The data type to be stored and managed was suited to be stored in a database, as they allow organized data storage, efficient data management (addition, removal, update and retrieval), and secure multi user access.

There are basically two kinds of databases: relational and non relational or NoSQL. Relational databases (RBMS or SQL databases), have a collection of tables of data items, all of which are formally described and organized according to a relational model. On the other hand, NoSQL databases are non-relational, distributed and horizontally scalable.

NoSQL database management systems are useful when working with large quantities of data and when the data's nature does not require a relational model. Even for data that can be structured, NoSQL databases can be useful for applications in which what really matters is the ability to store and retrieve great quantities of data, not the relationships between the elements. If the amount of data is large and relationships are needed, indexing and duplication of information in the stored documents are essential.

NoSQL databases are categorized according to the way they store the data and fall under categories such as key-value stores, document store databases, graph databases, multivalue databases, ordered key-value stores or key-value cache in RAM amongst others.

In our case, two conditions had to be taken into account when choosing a database. Streaming retrieval volume is not constant, ranging from 4 to 10 million tweets per day. Additionally we constantly retrieve the tweets of the selected

geolocalized users. We also note that significant political or social events may cause traffic spikes. Thus the database must have the capability and scalability to handle a large and steadily growing volume of data while providing enough margin and flexibility to cope with unexpected traffic peaks.

Apart from that, the individual messages or tweets are JSON [9] encoded. JSON, JavaScript Object Notation is a format used to transmit data in a human readable format (key, value) over a network connection as an alternative to XML. The attributes of a JSON-encoded object are unordered and not all the fields must appear in all the messages. In addition, JSON information contained in the tweets can change over time (fields can be added or removed) and the format of the field values (integer, string, datetime, ...) can also change.

Regarding scalability, which is one of our main considerations, for SQL databases this mainly relies on improving the server by adding memory, disk devices and/or cores. Even that in MySQL, master-master configuration allows writing to one and reading from the other, it was impossible to have a large percentage of data in memory to speed up queries.

Attending the scalability needs we focus on NoSQL databases out of which we consider CouchDB and MongoDB. CouchDB [4] uses JSON to store data and JavaScript as its queying language. Queries are basically map/reduce operations mapped in views, when adding a new query, a new view has to be added. It also provides ACID (Atomicity, Consistency, Isolation, Durability) [10] semantic and it does this by implementing a form of Multi-Version Concurrency Control, meaning that CouchDB can handle a high volume of concurrent readers and writers without conflict. Replication and failover are achieved by having multiple copies of the same data.

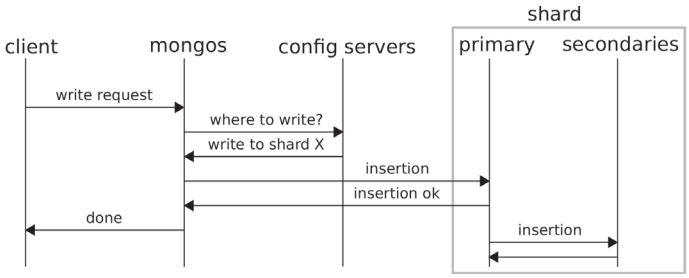

Fig. 1. MongoDB default write concern. MongoDB structure will be explained in section

On the other hand, MongoDB [5] uses BSON to store data (JSON-like documents with dynamic schemas). Supports SQL-like queries, map/reduce ones and aggregation, thus not restricting developers to a pre-defined set of queries. Indexing (up to 1024 indices per collection) allows queries to speed up and be run in realtime. Replication and failover are achieved the same way CouchDB does. Atomic transactions are only possible within the scope of a single document. But

attending to amount of concern the application has for the outcome of the write operation, durable writes can be achieved. With default write concern (Fig. 1), the application sends a write operation to MongoDB and the database confirms the receipt of the write operation. With stronger write concerns, write operations wait until MongoDB acknowledges or confirms a successful write operation. MongoDB provides different levels of write concern to better address the specific needs of applications such as confirm the write operation only after it has written the operation to the journal (it already implies durable operations) or after the write operation has propagated to the members of a replica set (group of computers with replicated data).

Both databases are quite similar and allow JSON documents. Even that MongoDB lacks real ACID transactions, it is more suitable when using large amounts of data. Horizontal scalability is much clear in MongoDB and the possibility to allow users to run their own queries without requiring additional configuration made MongoDB more suitable to our needs.

4 MongoDB configuration

In order to achieve high availability and scalability we use multiple replica sets. A MongoDB replica set, Fig. 2, is a cluster of mongo daemons (mongod) instances that replicate amongst one another and ensure automated failover. Replica sets consist of two or more mongod instances with one of these designated as the primary and the rest as secondary or delayed members. Clients direct all writes to the primary, while the secondary members replicate from the primary asynchronously with a delay of a few milliseconds, so that, reads can be performed either in the primary or the secondaries. Database replication with MongoDB adds redundancy, helps to ensure high availability and increases read capacity. Delayed members replicate the primary with a predefined delay time and we use them as backup devices.

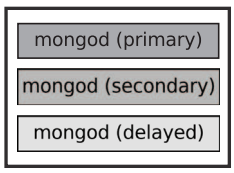

Fig. 2. Basic Replica Set

The current amount of data stored is more than 15GB of plain text per day, which represents an increase of 6TB per year in the size of the database. The approach to scale out, when one machine is not enough to store all the data, or write capacity needs to be increased, is sharding. It partitions a collection and stores the different portions (chunks) on different machines. Sharding

automatically balances data and load across machines and provides additional write capacity by distributing the write load over the computers. In addition to that, when a database collection becomes too large for the existing storage, a new machine (horizontal scalability) can be added and sharding automatically distributes collection data to the new server.

A sharded cluster consists of the following components:

- *Shards*. A shard is a container that holds a subset of a collections data. Each shard is either a single mongo daemon (mongod) instance or a replica set (RS)
- *Config servers*. Each config server (CS) is a mongod instance that holds metadata about the cluster. The metadata maps chunks or collection portions to shards.
- *Client instances (mongos)*. The mongos instances (CL) route the reads and writes from client applications to the shards. Applications do not access the shards directly.

Sharded MongoDB arquitecture requires a minimum of three configuration servers. The shards are usually replica sets and its number and internal structure depends on the amount of data to be stored and the reading speed needed for the applications. As for the number of client instances, this depends on the applications that need to access the database. We use just one client instance (CL).

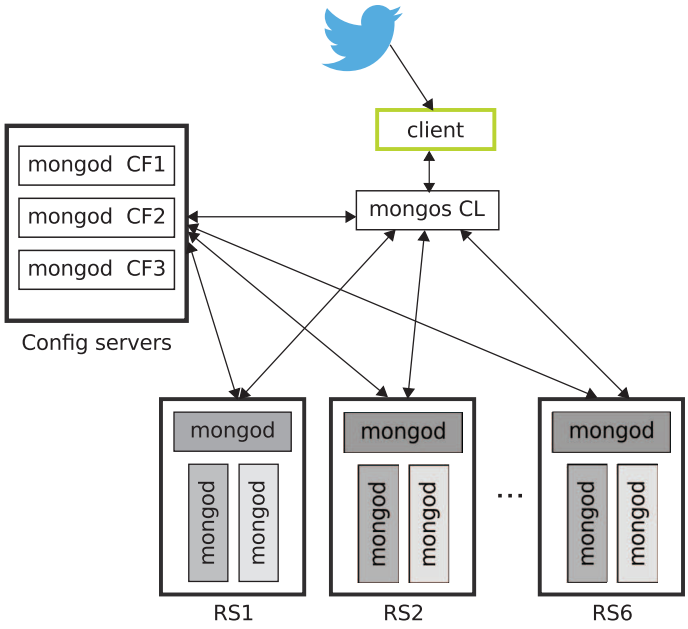

Fig. 3. Global MongoDB Configuration

In our case (Fig. 3), we use six replica sets with three members each. There are two elegible primary members and the third one is a delayed copy by 72 hours. This gives us failover security because if primary server crashes the secondary one move to primary status. And the third member helps us to recover from various kinds of human error such as inadvertently deleted databases or botched application upgrades.

The load of the configuration servers is small because instance maintains a cached copy of the configuration database and the total amount of activity is relatively low, therefore they are deployed as virtual machines with just one core and 1GB of RAM. The Client Instance (CL) also uses minimal resources and is also placed in a virtual machine alongside the application server and has two cores and 2 GB of RAM.

The shard key used is the tweet identifier and we added indices by user identifier and latitude/longitude to speed up usual queries.

To improve writing performance we took into account several MongoDB features when customizing the operating system in the servers that form the replica sets. One of the first considerations is that MongoDB uses write ahead logging to an on-disk journal to guarantee write operation durability and to provide crash resiliency. If the filesystem does not implements journaling and mongod exits unexpectedly the data can be in an inconsistent state. To avoid this issues we use ext4 since it implements journaling. Besides choosing a convenient filesystem, writing speed can be increased by mounting the file system where the database is located with the option noatime avoiding the logging of the record of the last time the file has been accessed or modified.

Apart from the filesystem configuration, some system parameters can be tuned for better write/read performance. In particular, the number of open files was increased to 96000 from the 1024 default value and the number of processes or threads per user to 64000 (in the replica sets all the MongoDB activity is handled by the user mongodb). We also avoided using hugepages related to NUMA kernels.

Finally, we note that the write concern policy we are using is the default one, see Fig. 1, because insertion speed in our case is more important than assuring that every single tweet correctly stored (the eventual loss of a single tweet is of little relevance for the study of the overall mobility patterns).

5 Results

We compared MySQL and MongoDB insertion speed. In MySQL we used Django Object Relational Mapping, ORM, to map JSON to objects. Django ORM is one of the most used Python object relational mapping alongside SQLAlchemy. In MongoDB we just inserted the JSON objects with no preprocessing.

5.1 Insertion performance

Figures 4 and 5 compare the results for the time it takes to store 100000 tweets in the database using a MySQL server and the MongoDB system described above with three replica sets.

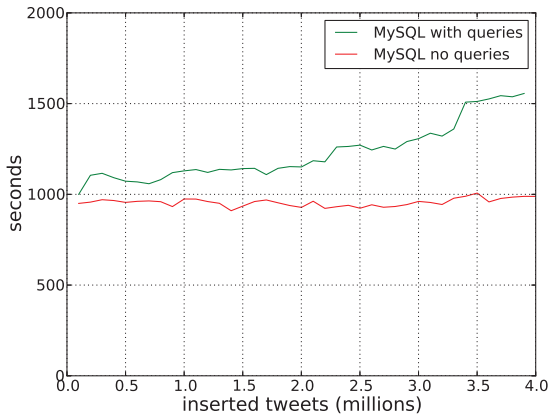

Fig. 4. Time to insert 100000 tweets in MySQL using an empty database and tweets processed with ORM. Linking (green) and duplicating information (red)

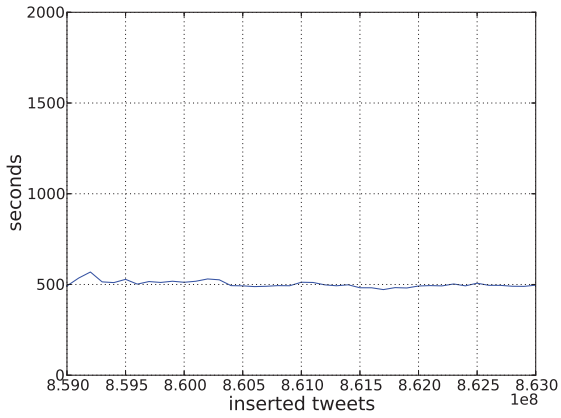

Fig. 5. Time to insert 100000 tweets in MongoDB by using direct insertion in a database with millions of tweets.

In Fig. 4 MySQL data is shown starting from a completely empty database. When inserting tweets in MySQL, as it is a relational database, we first perform a search to find if the twitter user exists, if not, a new record is created, while if the user is already there, a link to the existing register is performed. This requires a search for every tweet to be inserted which results in a larger storage time and in the fact that as the database grows the search takes longer and the insertion rate decreases. It takes 1000 s when it is empty, above 1500 s when there are four million tweets and almost 4000 s when the database has twelve million tweets.

Then, taking into account that in MongoDB no queries are done when inserting tweets and duplicated information is stored, we tested the same in MySQL, even that it means not using the relational properties of a relational database such as MySQL. In this case we just used the ORM to convert from JSON to MySQL objects. This is further illustrated Fig. 4 and shows that the injection rate remains almost constant over the whole range of 4 million tweets, just showing a minor reduction when the number of tweets increases.

In MongoDB, Fig. 5, on the contrary, tweets are inserted without searches and finding if there are tweets of the same user is done upon client request as part of the search query. Therefore the storage time is much smaller, around 500 s for the 100000 tweets, which is a speed up factor two with respect to the MySQL when no search is performed. Although the speed up is smaller than the factor three expected from the fact of having three replica sets, it is still substantial. What is more important, since we do not need to perform searches, this performance is maintained as the database size grows and in Fig. 5 shows the performance after the insertion of 850 million tweets.

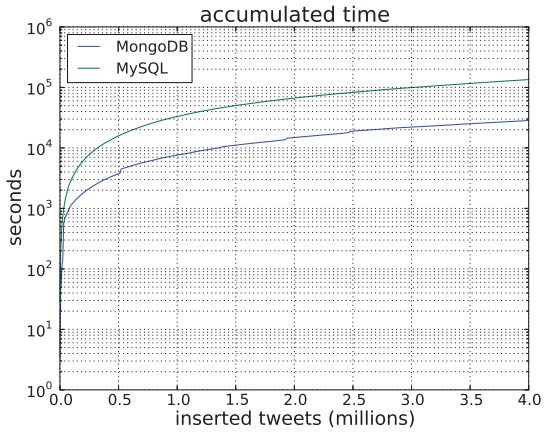

Fig. 6. Database insertion comparison, accumulated time

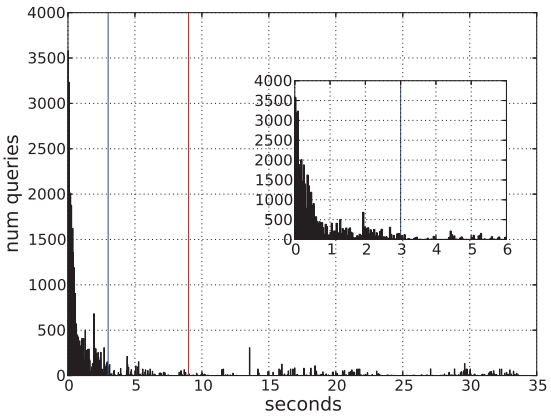

Fig. 7. Queries timing histogram for Barcelona metropolitan area. Blue line shows the median and red line the 70th percentile.

Finally, Fig. 6 shows the accumulated time in logarithmic scale to insert four million tweets in an empty MySQL database with relational queries and MongoDB. The difference in accumulated time grows exponentially as the database size increases.

5.2 Query performance

We are mostly interested in the fraction of the tweets stored in the database that are geolocalized, therefore MongoDB spatial indexing and geospatial commands play an important role.

MongoDB offers a specific geospatial index 2d for data stored as points on a two-dimensional plane. As of version 2.4 MongoDB also includes the index 2dsphere which conveniently supports queries that calculate geometries on a sphere. This index supports data stored as GeoJSON objects [11] which is the way geospatial data is stored in the tweets. Despite that, since we started with MongoBD 2.2, we are currently using 2d indices (latitutde, longitude) to determine the localization of the used when the tweet was posted.

MongoDB also includes the command geoNear which returns the documents on the database which have a geospatial location closer to a given location. The geoNear command can be used either with 2d data as well as with GeoJSON objects. In Fig. 7 we can see the histogram of geoNear queries in a database with one thousand million documents. We made use of a radius of one mile and a fine grained mesh over Barcelona metropolitan area in which the points were separated by one mile.

Even there is a group of slow queries, thirty seconds or more, the median is just of three seconds, and in 70% of the queries to get the tweets localized in a radius of one mile of a given point lasted less than nine seconds.

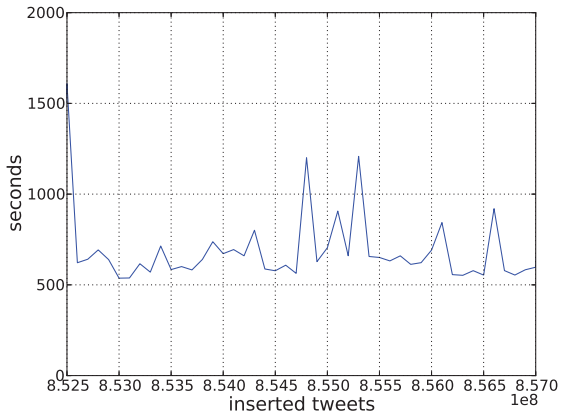

Fig. 8. MongoDB insertion with a database with millions of tweets while querying the DB with CPU and memory consuming geoqueries.

Finally we address the influence of the queries in the insertion data rate. In Fig. 8 we show the insertion rate in MongoDB when simultaneously performing queries on a database which has already stored 850 million tweets, so that queries require searching over a non trivial amount of data. The structure of MongoDB allows the queries to be performed on the primary nodes or on the secondary ones. The most disturbing situation for the insertion rate is when queries are performed on the primary nodes, and this is the case shown in Fig. 8. The presence of the queries induce peaks in the time to insert 100000 which can go over 1000 s but nevertheless the overall response of the system is quite satisfactory and the performance sufficient to keep storing all the tweets. Fig. 8 shows, in fact, the worst-case scenario. In practice, queries are performed over secondary nodes and in that way the insertion data rate is practically unaffected.

5.3 Preliminary results for mobility patterns

Preliminary results after retrieving data for six months show that London and Barcelona commuting areas are well defined just by using geolocalized tweets, see Fig. 9. A visual inspection shows that geolocalized data is distributed as the population density for the different cities, which means that already the six months sampling is representative. In order to further assess that the data is statistically adequate we plan to compare the statistics obtained from this six months retrieval with the ones obtained after one year.

Finally, in the framework of EUNOIA project public transport data and Twitter data amongst others will be used to characterise and compare mobility and location patterns in different European cities.

6 Concluding remarks

In summary, we have presented an example of big data retrieval and efficient data storage based on a no-SQL database, MongoDB. We have seen that geolocalized queries are suitable for large datasets and are not time consuming. Besides, we have been retrieving the maximum amount of geolocalized data from Twitter just taking into account some cities where mobility patterns are currently being studied.

Because NoSQL databases have weaker data consistency models, they can trade off consistency for efficiency and stand out in speed and volume. As for this, applications that need to use large amounts of data, data that grows over the time, schemaless data or geolocalized data are suitable to use MongoDB as storage.

Acknowledgements

Financial support from CSIC through project GRID-CSIC (Ref. 200450E494), from MINECO (Spain) and FEDER (EU) through projects FIS2007-60327 (FISICOS) and FIS2012-30634 (INTENSE@COSYP) and from European Comission through project FP7-ICT-2011-8 (EUNOIA) is acknowledged.

References

1. Douglas, Laney. "3D Data Management: Controlling Data Volume, Velocity and Variety". Gartner. 6 February 2001. http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Contr olling-Data-Volume-Velocity-and-Variety.pdf
2. Douglas, Laney. "The Importance of 'Big Data': A Definition". Gartner. Retrieved 21 June 2012. http://www.gartner.com/resId=2057415
3. Twitter web page. http://www.twitter.com
4. CouchDB web page. http://couchdb.apache.org

Fig. 9. Geolocalized tweets

5. MongoDB web page. http://www.mongodb.org
6. geoNear documentation. http://docs.mongodb.org/manual/reference/command/geoNear
7. BSON document size limitation. http://docs.mongodb.org/manual/reference/limits
8. GridFS. http://docs.mongodb.org/manual/core/gridfs/
9. JSON. http://www.json.org/
10. ACID. http://en.wikipedia.org/wiki/ACID
11. GeoJSON. http://geojson.org/geojson-spec.html